IGOR:
The Intelligence Guard for ONI Replication

R.W. Shore

The ISX Corporation
2000 North 15th Street, Suite 1000
Arlington, VA 22201
703/558-7800 (V), 703/558-7895 (F)
bshore@isx.com

Abstract: The Intelligence Guard for ONI Replication (IGOR) is a dual-host guard processor that allows database replication to occur across a security barrier with no person in the loop. IGOR works by accepting and validating SQL statements passed to it from a Sybase Replication Server (a product of Sybase, Inc.). A validated SQL statement flows across a serial line to the "other side" of the security barrier, where it is applied to the replicate database. IGOR's configuration files describe the SQL statements that are allowed to flow across the security barrier, including value checks that must be applied to validate the statements. Each of the two hosts associated with an IGOR installation is dedicated to processing SQL statements; only a limited number of UNIX users with well-defined roles are allowed to login to an IGOR host. IGOR has been accredited for a specific high-to-low installation. With different configuration files the same code can be used for other high-to-low situations, and with minor additions to the code IGOR would be appropriate for low-to-high situations as well.

This work was funded by the Office of Naval Intelligence, National Maritime Intelligence Center, 4251 Suitland Road, Washington DC 20395-5020. The Government point of contact is Mr. Al Poulin, 301/669-4000.

1. Introduction

One problem facing the Office of Naval Intelligence (ONI) is the dissemination of its analytical databases to customers at various security levels, in a timely and secure fashion. In the commercial world the technology of database replication is becoming one mechanism for keeping two (or more) copies of the same database synchronized automatically. Before this technology could be applied to ONI's problems, however, we had to develop a security guard that would allow the automated replication process to occur in a secure and controlled fashion.

This paper describes IGOR, the result of a nine-month effort by ONI to take advantage of the commercial replication software without compromising the security of ONI databases. The paper briefly discusses IGOR's operation and security features.

1.1. Glossary

GP
: Guard Processor. A role an IGOR host may play. The other role is the RSP.

IGOR
: The Intelligence Guard for ONI Replication. A GOTS product that allows a replication server to operate across a security barrier. Each IGOR installation consists of two hosts connected via a serial cable.

LTM
: Log Transfer Manager. Software that transfers changes from a master database into a Sybase replication server.

ONI
: The Office of Naval Intelligence, located in the National Maritime Intelligence Center (NMIC) in Suitland MD.

Replication Server
: A COTS product from Sybase that synchronizes a replicate database with a master by passing changes from the master to the replicate..

RSP
: Replicate-Side Processor. A role an IGOR host may play. The other role is the GP.

SQL

Structured Query Language. A near-standard syntax for expressing database changes.

1.2. Summary of IGOR's Operation

A replication server sends an SQL statement through a TCP/IP-based network to the host fulfilling the GP role. The GP verifies that the contents of the statement are in accordance with the security policy; specifically, the GP verifies that the statement mentions only the expected database, tables, and columns and that columns pass any value constraints given in the security policy. If the statement passes the checks, the GP passes the statement across a serial line to a second host fulfilling the RSP role. The RSP applies the statement to the target database via a second TCP/IP network and the appropriate, DBMS-specific protocol. The RSP returns a pass/fail status back through the serial line to the GP, which passes the status to the replication server.

IGOR's initial accreditation involved a high-to-low transfer, with the GP connected to an SCI network and the RSP connected to a non-US SECRET-level network; this is the mode discussed in the bulk of this paper. With relatively minor additional protections on the RSP side, IGOR appears accreditable for low-to-high operation. It is technically possible for each of a pair of IGOR hosts to fulfill both the GP and RSP roles, although there are no current plans to accredit IGOR in this mode.

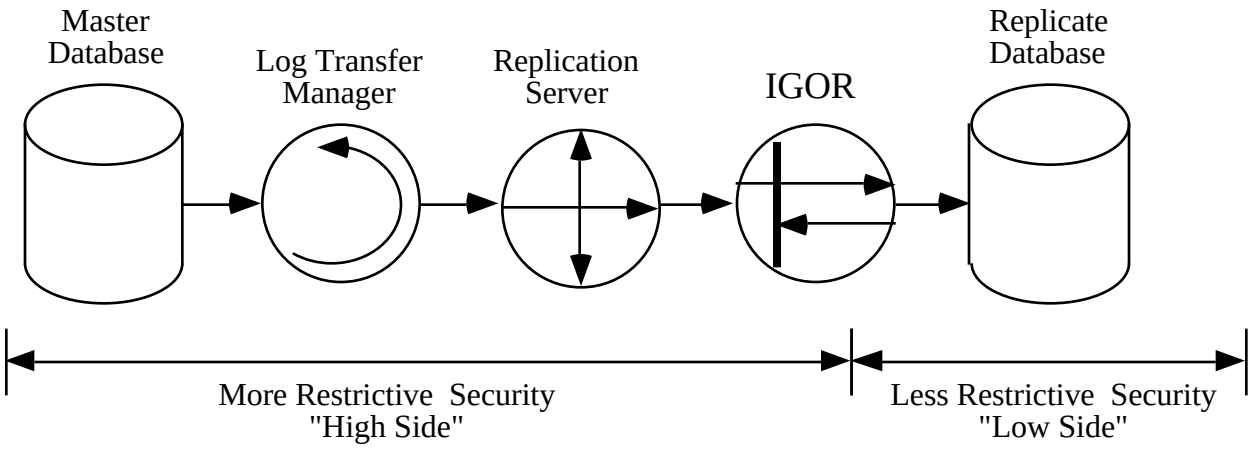

Exhibit 1 Basic IGOR Architecture

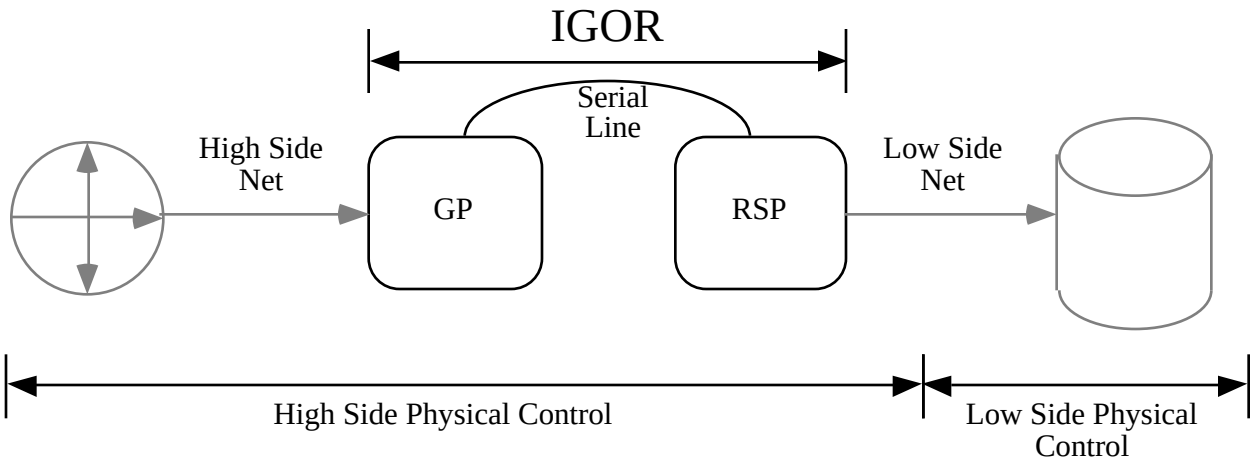

Exhibit 2 IGOR Hosts

2. IGOR's Operation

Exhibit 1 indicates the overall environment within which IGOR is to work. The following entities appear in Exhibit 1:

- Master Database: The source of changes are tables in a designated "master" database which may be managed by either Oracle or Sybase. Each change to a replicated table flows to a …

- Log Transfer Manager (LTM): An LTM is an intermediary responsible for passing changes from a master database to a …

- Replication Server: Within the replication server, changes to the master are queued and eventually distributed to one (or more) replicates. Exhibit 1 shows only one of an indefinite number of replicates, not all of which must involve IGOR. In the depicted situation the replication server passes each change to …

- IGOR: IGOR verifies that the change being passed from the replication server is appropriate for this specific replicate. If the change is proper, IGOR simply applies it to the …

- Replicate Database: This database contains a copy (perhaps a subset) of the master. The replicate can be managed by either Oracle or Sybase; the replicate's DBMS need not be the same as the master's.

The result of this process is that changes made to the master are also made to the replicate, in near real time and without a person in the loop.

Exhibit 2 shows a slightly expanded version of the IGOR instance appearing in Exhibit 1. As depicted in Exhibit 2, IGOR consists of two processors connected via a serial cable. The GP is connected to the same network as the master database; the RSP is connected to the same network as the replicate.

The replication server connects as a client to the GP and passes all changes to it, where a "change" takes the form of an SQL statement. For each row in the master that changes, the replication server emits one of the following SQL statements:

- `insert`: indicates that a new row has been added to a specific replicated table in the master.

- `delete`: indicates that a specific row has been removed from a specific table in the master.

- `update`: indicates that a specific row in a replicated table has changed.

Each SQL statement carries additional information to completely specify the change. For example, an `insert` statement includes the name of the replicated table, the names of all the columns, and the value associated with each column.

It should be noted that the replication process is a near real-time duplication of changes to the master. There is no filtering or consolidation of redundant changes at any step of the process. Suppose, for example, that a single transaction adds a row to a replicated table (`insert`), changes some values in that row (`update`), and then removes the row (`delete`). There is no net change to the master database. Even so, the replication process would faithfully reproduce this same sequence of SQL in the replicate database.

The replication server itself is a COTS product of Sybase Inc., which means that IGOR must live with the benefits and liabilities associated with COTS. Among the benefits is the fact that the replication server is a highly asynchronous operation.

- Changes move to the LTM only after the underlying database transaction has passed a commit point; the master's throughput is not seriously impacted by the replication process.

- The LTM uses either the transaction log (Sybase master) or trigger-maintained change-description tables (Oracle and other masters); while the LTM is processing a transaction, the update software running on the master is not blocked by the LTM's activities, nor are there any particular requirements levied on the master's maintenance software to support replication.

- The replication server accepts a change from an LTM by placing it into a disk-based "stable queue". Once the replication queues a change, the LTM is free to remove it from the master's tables or logs; barring a catastrophic disk failure, the replication server guarantees to deliver each queued change eventually.

- The replication server delivers changes to a replicate at the speed of the replicate, not the speed of the master. This is particularly

important for IGOR, which is often limited by the speed of the serial line. Of course, the average change rate in the master cannot exceed the capabilities of IGOR; otherwise the replication server's queues will eventually fill.

As indicated by the preceding discussion, IGOR is the replicate database as far as the replication server is concerned. That is, the replication server logs into IGOR, passes SQL to IGOR, and gets success and error status returns, just as if IGOR were a Sybase SQL Server managing the replicated database directly.

From IGOR's perspective, of course, the situation is quite different. When it receives an SQL statement from the replication server, IGOR performs the following checks on it.

1. IGOR completely parses the statement. A statement that IGOR does not recognize as a proper `insert`, `delete`, or `update` generates an immediate error back to the replication server with nothing passed from the GP to the RSP.

2. IGOR verifies that name of the table appears in the statement and that the table is one IGOR expects to be replicated. If the table name is missing or incorrect, IGOR generates an error to the replication server and again passes nothing.

3. IGOR verifies that the statement names all columns and that each column is one IGOR expects. If the statement contains an unexpected column name or "anonymous" data (values not explicitly connected to a named column), IGOR generates an error back to the replication server without passing the SQL to the RSP.

4. For an `update` or `delete` statement, IGOR verifies that the `where` clause specifies a value for each primary key and that only primary-key columns appear in the `where` clause. If a key is missing or if the `where` clause contains an unexpected column, IGOR generates an immediate error back to the replication server without passing the SQL to the RSP.

5. For each columns with a value constraint (discussed subsequently), IGOR verifies that the value mentioned for that column is acceptable. If a column's value is bad, IGOR generates an immediate error to the replication server and passes nothing to the RSP.

6. If the table is subject to multi-table filtering, then IGOR runs the appropriate SQL and checks the return value. The need for multi-table filtering is discussed later in this section. If the change fails a multi-table filter, IGOR sends a "success" status to the replication server without passing the SQL to the RSP.

7. If the table is subject to full verification and the change is an `insert` or `update`, IGOR verifies that the row in question actually exists in the master database. If the row does not exist, IGOR sends a "success" status to the replication server without passing the SQL to the RSP.

If and only if the statement passes all of these checks, IGOR rebuilds the SQL from the representation generated by check #1 and passes the reconstituted statement to the RSP, which applies it to the replicate and passes a pass/fail status back. The GP sends the status to the replication server.

It should be noted that one of the reasons IGOR can reliably validate the statements passed from the replication server is that the server generates predictably-formatted SQL statements within a small subset of full SQL. For example, the server never generates a `select` statement, which is one of the more complex SQL statements to parse. Furthermore, the replication server's internal workings guarantee that each SQL statement describes a change to precisely one row of the master database. This is why check #4 above makes sense; each change must pick exactly one row by specifying a value for each primary-key column.

The checks that IGOR makes on each SQL statement can be divided into syntactic checks (#1–#4) and content checks (#5–#7). The syntactic checks verify that the statement is well-formed and mentions only the expected database, rows, and columns; these will be discussed no further here.

The value check (#5) ensures that the values in specified columns are in accordance with the security policy. IGOR allows the security policy to specify at most one wild-card expression (one UNIX regular expression) for each column; IGOR passes only values that match the expression. IGOR applies this check to the `values` clause of each `insert` statement to the `set` clause (not the `where` clause) of each `update`. For example, the security policy for IGOR's initial accreditation specified the following two restrictions:

- The value for column X in table T must be `M` or F. The associated regular expression is `'[MF]'` (the quotes are part of the expression).

- The value for column Y in table T must start with either M or N. The regular expression is `'[MN].*` (the quote is part of the expression).

Note that within the scope of check #5 IGOR examines a column only if it has a declared, specific UNIX regular expression. In particular, IGOR makes no attempt to do a generic "dirty value search" through all the columns being passed from the GP to the RSP; IGOR limits its checking to those columns constrained per the security policy.

Multi-table filtering (check #6) requires a bit more motivation. Consider, for example, a hypothetical high-side master database of aircraft locations (`ac_db`), and suppose that a low-side replicate needs to have only aircraft produced by a specific list of countries (`A`, `B`, `C`) passed to it. A typical database design for the master would put all the fixed information about aircraft (including the producing country `p_ctry`) in one table (`ac`) and the current location of the aircraft in another (`loc`); a key, such as a randomly-generated aircraft identifier (`ac_id`), indicates which rows in the location table are associated with which row in the aircraft table.

Now consider IGOR's dilemma when it is handed a change from `loc`. The security policy says that only aircraft produced by certain countries can be passed to the replicate, but a row from `loc` does not contain `p_ctry`. The only way that IGOR can decide whether or not to pass the change to the replicate is to consult `ac`, back in `ac_db`. The term "multi-table filtering" denotes that fact that IGOR needs information from other table(s) to determine the suitability of a particular row for the replicate.

In some cases it is possible to avoid the need for multi-table filtering by changing the database design. In the example above, for instance, the need for multi-table filtering disappears if we simply add `p_ctry` to `loc`. Database redesign is not always possible:

- Placing redundant data in tables is generally viewed as bad technical design and is often resisted by system analysts.

- Changing the structure of an existing production database and its maintenance software can be a long and expensive (thus undesirable) process.

When IGOR was being designed, it seemed prudent to implement multi-table filtering. There is a cost associated with multi-table filtering: for each change from a table subject to multi-table filtering, IGOR must validate the change by issuing a `select` statement back to the master database and checking the return value. This increases the transaction load on the master and may nearly double it if the table has a high rate of change.

It turns out that "multi-table" filtering can also be used to implement complex checks that cannot be handled by IGOR's simple column-by-column regular expressions. Suppose (to continue the hypothetical aircraft example) that IGOR is supposed to pass fighter aircraft produced in `A`, `B`, or `C` and transport aircraft produced in `X`, `Y`, and `Z`. The replication server can perform this sort of filtering (from a technical perspective, this constraint is an "or" of two "and"ed conditions), but IGOR cannot use its value checks to verify the replication server's filtering; IGOR's value-checking implementation does not support this sort of cross-column constraint. However, a "multi-table" filter that references only the `ac` table can be constructed so that IGOR will enforce this constraint. Again, however, the multi-table filter carries the penalty of a higher transaction load on the master.

Full validation (check #7) tells IGOR to ensure that each passed row actually exists in the master database. This check is very expensive in terms of the increased transaction load on the master and is not currently planned for use at ONI. It would be appropriate only when the master database is extremely sensitive and/or there appears to be a need for additional protection against uncontrolled, "rogue" programs attempting to use IGOR's facilities.

3. IGOR's Installation

The first requirement for an IGOR installation is a written security policy that specifies precisely what information is allowed to flow from the GP to the RSP and that is approved by (1) the owners of the information in the master database and (2) the appropriate security authorities. In technical terms, the security policy must be specific enough to specify a view of each replicated table. IGOR's basic job is to ensure that only the allowed view of each replicated table passes from the GP to the RSP. Some of the considerations associated with an IGOR installation appear in the subsequent sections.

3.1. Configuration Control

IGOR performs no queuing or other storage of SQL statements or database contents. All queuing occurs in the replication server; IGOR is a pass-through operation only. Thus except for deliberate maintenance activities (see Section 3.3 below), IGOR expects the content and location of most files to be static. To protect the configuration, the following features exist on both IGOR hosts.

1. IGOR runs with the keyboard disconnected and with no unnecessary peripherals (such as a CDROM drive) connected. This makes it more difficult to access the hardware console to perform a single-user boot.

2. After IGOR is installed, the superuser `root` is locked out. There is no way to gain interactive superuser status on an IGOR host; special IGOR setuid `root` applications provide limited `root` access to the UNIX logins on an IGOR host.

3. Most standard UNIX demons are not started. NFS and `sendmail`, for example, do not run. The only background process spawned by the `inetd` process is `telnet`; `ftp`, `finger`, and other such processes are not available.

4. Each time it starts, IGOR computes a file signature for critical configuration files and directories and compares the computed signature with a stored signature. If there is a mismatch, IGOR refuses to run.

Together, these features mean that it is difficult to change IGOR's configuration, and if an unexpected change does occur, IGOR shuts down the SQL transfer process. There is a back door to the IGOR host: a boot from a CDROM or other alternative media will allow an administrator to achieve single-user status and to unlock the `root` password so that interactive `root` access is possible. The absence of the keyboard and CDROM drive on the IGOR host during normal operation means that an alternative-device boot is a relatively complex and public process. Thus only a maintainer with proper authorization is likely to have access to the IGOR hardware for sufficient time to unlock `root`.

3.2. Access Control

As discussed in subsequent sections, IGOR includes two distinct access-control concepts: access via UNIX mechanisms and access via IGOR itself.

3.2.1. UNIX Access Control

Access to an IGOR host via UNIX mechanisms is limited by the following considerations.

- As mentioned previously, `root` is locked out. There is no way to achieve superuser status without an alternative-media boo.

- There are only two authorized userids on an IGOR host, conventionally called `igoradm` and `igorisso`. These userids have well-defined roles, as discussed in Section 3.3. All other userids in the password file are locked out at installation time.

- With most standard demons disabled, the only way one can access an IGOR host via UNIX is via `telnet` through the network.

Since `root` is locked out, there is no mechanism by which anyone can define a new userid. IGOR does include a setuid `root` module that allows a manager to clone a new administrator or ISSO, as discussed in Section 3.3. However, from a UNIX perspective a clone is identical to either `igoradm` or `igorisso` rather than being a separate and independent user.

3.2.2. IGOR Access Control

IGOR's GP is server software to which the replication server connects as a client. The GP has its own set of authorized userids and passwords, independent of the UNIX password file. IGOR recognizes two general classes of userids:

- An "incoming" userid is one that the replication server or other client uses to connect to IGOR.

- An outgoing userid is one that IGOR uses to connect to an external server. For example, IGOR needs an outgoing userid to connect to the replicate database and update it.

The passwords for these userids appear in IGOR's configuration files as encrypted values. For incoming userids, IGOR uses the same concept as UNIX to

store passwords: the password field contains a value for which the clear-text password is the decryption key. Until an external user supplies the password to IGOR, the GP does not have the information it needs to decrypt the password field.

For the password for an outgoing userid, IGOR uses the fact that each incoming userid is associated at accreditation time with a single replicate database and set of verifications checks and hence with a fixed set of outgoing userids. IGOR stores the passwords for outgoing userids in encrypted format, using the clear-text *incoming* password as the decryption key. Thus IGOR needs the incoming password not only to validate access by a specific incoming userid but also to decrypt the necessary outgoing passwords.

Since IGOR uses the COTS OpenServer library from Sybase and expects connections from Sybase's replication server, any additional access control checks that IGOR might implement are constrained by the features provided by these two products. In particular, IGOR cannot reliably determine the host from which a connection is coming (the OpenServer does not provide this information) and cannot use any authentication scheme (such as a challenge-response sequence) beyond a simple password check (the replication server does not support any other scheme). This means that there is at least a theoretical possibility that an agent other than the replication server will attempt to connect to IGOR using the replication server's userid and password. IGOR implements the following obstacles to such an attack:

- The attack would have to come from the GP-side network. A user on the RSP network has no access whatsoever to the GP; even if the RSP is totally compromised the serial line between the GP and RSP uses a customized, IGOR-only protocol that provides no access to the GP's TCP/IP network.

- IGOR allows only one active connection for each incoming userid, and the replication server is generally connected to IGOR at all times. This limits IGOR's vulnerability window.

- If an attempt is made to open a second connection with an in-use userid, IGOR refuses the connection, shuts down the existing connection, and disables the userid. IGOR refuses all subsequent connections under that userid until the IGOR code restarts, either as a result of a reboot of the host or an administrative shutdown command to IGOR itself.

- The passwords associated with incoming and outgoing userids expire at an interval defined in IGOR's static configuration file, which is fixed at accreditation. This limits the length of time that an appropriated password will be valid.

- IGOR can be configured to verify that each row passed to the RSP is in fact in the master database. IGOR makes this check in addition to value checks and multi-table filtering. This option is very expensive in terms of the transaction load imposed on the master database, however, and is not currently used at ONI.

3.3. Maintenance

In general, IGOR maintenance follows a two-person rule: `igoradm` proposes and `igorisso` validates. Specific maintenance concepts include:

Database configuration: IGOR allows table names, column names, allowed values, and other parameters for SQL validation to change as the master database and security policy evolve. IGOR allows `igoradm` to propose a complete replacement for the database configuration file that describes the allowed SQL. `igorisso` must approve the replacement file (without change) before IGOR will actually use it. IGOR allows tables, columns, and so on, to change but does not allow a new database or database server to be added as either a master or replicate; server names appear in the static configuration file which is fixed at accreditation.

User configuration: IGOR provides a special application that `igoradm` and `igorisso` use to manage IGOR's incoming and outgoing userids. `igoradm` can add and remove entries from the user configuration file; `igorisso` can initialize and change passwords in existing entries. Note that a new userid cannot be employed until `igoradm` makes an entry in the configuration file and `igorisso` initializes the password.

Clone UNIX users: IGOR expects that there may be multiple individuals that can play either the administrative or ISSO roles and thus need UNIX passwords on an IGOR host. To help manage the UNIX passwords, IGOR provides a module that can clone either `igoradm` or `igorisso`. A clone for

`igorisso` (for example) has a separate entry in the UNIX password file but runs under the same numeric userid as `igorisso`. A clone is simply an alternative password and is not an independent userid. `igoradm` (or any of its clones) can create a new clone; the clone is not usable until `igorisso` (or any of its clones) assigns an initial password. `igoradm` (or any of its closes) can remove a clone.

3.4. Alerting and Logging

IGOR was designed to run without a human operator and without human intervention most of the time. To keep its managers informed of various internal conditions, IGOR uses the UNIX mail system to send alerts to addresses outside of the IGOR hosts; although `sendmail` is disabled for incoming mail, IGOR can still send mail to external hosts. IGOR sends mail to an arbitrary number of addresses (specified in the database configuration file) whenever it starts up, whenever a serious error prevents IGOR from running, and whenever other "interesting" situations occur.

IGOR maintains a log of important events (UNIX login, IGOR login, and the like) and (on the GP) a complete list of all SQL statements sent to the RSP. A timed batch job (`cron` job) archives these logs, as well as other UNIX-maintained log files, to tape. The archive script `Ssalvage` checks for various error conditions during the archive run and alerts managers (via mail) when a tape needs to be replaced, the archive run fails, or other error conditions. Other than periodic replacement of a full archive tape, IGOR runs completely automatically, with no operator intervention required.

4. Status and Future Work

IGOR currently runs on two Sun Sparc IPX platforms running standard Solaris 2.4. Due to hardware limitations on the IPXs, IGOR's serial I/O is limited to 9600 baud. Two tables are being replicated from an ONI production database, one with approximately 20 attributes and the other with approximately 70. IGOR is handling about 15,000 inserts and 20,000 deletes per day, and could possibly handle as much as twice that load before the serial port bottleneck becomes critical. The CPU load on the system is low (generally under 20%); IGOR is definitely an I/O-bound process.

IGOR is built with Sybase's OpenServer product. It is multi-threaded; it can handle multiple connections and allows multiple tables to be replicated through a single connection. A design limit constrains each connection to involve a single source database and a single replicate database; the IGOR userid employed for the connection uniquely determines both the master and replicate databases, per the IGOR configuration files. Note that each IGOR installation is a guard between two specific security environments, one associated with the GP's network and the other associated with the RSP's network. Each distinct pair of security environments requires a separate IGOR installation.

During the week of 11-March-1996 this IGOR installation underwent accreditation tests by a team consisting of representatives from ONI-5 and DIA. The test uncovered no Category-I findings for IGOR itself (the only Cat-I finding involved accreditation for the master database). Of the two Category-II findings, one involved a minor code change and the other called for changes to IGOR documentation. There were several lower-category findings as well. All these findings have been resolved.

As mentioned in Section 1, the accredited IGOR installation operates in a high-to-low mode. Additional high-to-low IGOR installations would first require a written and approved security policy. The IGOR configuration files would next be constructed in accordance with this policy. The site would need to create installation-specific documentation, as an annex to the existing IGOR documentation, that describes the concept of operations and IGOR operational policies for the specific installation as well as a few security tests that depend on the structure of the master database. Finally, a security review or accreditation would be necessary to verify proper installation of the IGOR code on the new hosts and proper implementation of the security policy in the IGOR configuration files.

It is probable that low-to-high accreditation will require minor changes in the IGOR code. As implied by the other discussion in this paper, all of IGOR's validation activities currently take place on the GP. In the case of low-to-high replication, the GP roll is played by the IGOR host on the *low* network. Even though the GP itself is under the physical control of the high-side environment, there is at least a theoretical possibility that the GP could be compromised and all its protections removed. The following suggestions for IGOR changes to deal with

low-to-high issues are proposals by the author; they are not sanctioned by ONI, nor have they been seriously discussed with any accreditation authority.

One protection that is clearly needed for low-to-high operation is a limit on the databases that the RSP can access. IGOR currently stores all access information on the GP side; the GP passes the name of the database and the userid/password to the RSP through the serial line. This approach is satisfactory for high-to-low operation; for low-to-high operation, the RSP should have its own table of allowed databases. This change is a fairly simple one in the RSP code. With the change in place, the RSP ignores the database identification passed from the GP. Instead, it uses the userid from the GP to look up the database and the real userid; the password from the GP is the decryption key for the real password. This approach not only limits the databases to which the RSP can connect, but also shields the real database name, userid, and password from the low-side IGOR maintainers. In addition, the DBMS privileges associated with the RSP's database userid can ensure that only the proper subset of the replicate database is visible to the RSP.

The RSP should also implement some SQL checks for low-to-high operation. For example, the RSP should verify that the SQL coming from the GP always has certain primary key fields with specific values; this check would ensure that data coming from the low side is properly marked and cannot be confused with similar data that originates on the high side. These checks are a fairly simple extension of the existing SQL parsing and checking capability already used on the GP side.

It was noted in Section 1 that the two IGOR hosts can theoretically play both the GP and RSP roles. That is, the situation might arise in which databases on the low-side databases need information from the high side and also high-side databases need information from the low side. The only code module in common between the low-to-high and high-to-low information flow in this situation is the serial-line handler, which has very limited functionality and thus can be verified to work properly without a great deal of work. At this point it seems that a dual-mode operation is feasible from a security perspective, although there are no plans to actually accredit any IGOR installation for this sort of operation.

Finally, the reader will note that IGOR was built on standard Solaris. This situation exists mainly due to the lack of any approved product that would support Sybase's OpenServer and OpenClient libraries as well as Oracle's OCI library, but this situation will of course change over time. In a full multi-level environment with an approved, trusted operating system, IGOR's activities become that of a set of modules allowed to perform a specialized reclassification operation (write-down for high-to-low operation, write-up for low-to-high operation). In this environment, most of the SQL checks that IGOR currently performs would still be required. That is, IGOR would still have to verify that the SQL statements are appropriate for the RSP's security environment unless there are radical changes in the trusted versions of Oracle and Sybase and also in the replication server. Without such changes, most of the existing considerations, including multi-table filtering, would still apply. Although parts of IGOR's code would require modification for the new environment, much of it should port with little or no conceptual change.